Großer Beleg

Porting the VMKit framework and the J3 Java Virtual Machine to L4Re

Marcus Hähnel

September 5, 2011

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Björn Döbel
	Dipl.-Inf. Adam Lackorzynski

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den September 5, 2011

Marcus Hähnel

Danksagung

(**Todo:** *last but not least*) Todo

Contents

List of Figures

Listings

1 Introduction

The Java programming language [AGH05] that has been around since 1995 has seen a wide and fast adoption. Besides its powerful language features and the versatile frameworks available, platform independence plays a big role in its continuing popularity. Virtual machines are one of the factors that guarantee this independence of the operating system and the basic machine architecture, on which the code should run. To achieve this, the code is not compiled into a native binary, but instead into an intermediate format called *bytecode*. This format is well defined [LY99] and its endianness and other features do not depend on the platform on which the bytecode is compiled or executed. When it is executed this bytecode needs to be transformed into instructions the architecture understands. This transformation is either achieved by interpreting the bytecode or, for performance reasons, compiling it to the native platform either at the start of the program (Ahead-of-Time / AOT) or on demand with a so called Just-in-Time (JIT) compiler that uses on the fly binary translation.

Since the success of Java, more virtual machines, and languages that are compiled to them, have been developed, such as the .NET platform called Common Language Runtime (CLR) [MWG00] that is based on the standardized Common Language Interface (CLI) [Int06] with its most prominent language C#, or the new Parrot VM [Fag05] that was developed for Perl6 but is intended to be a more universal platform that more languages can target.

Because it is usually not possible to run bytecode directly on native platforms, it must be executed in another way. One way to run such a program is using an interpreter to execute the bytecode, but this means that for each instruction of the virtual machine many instructions on the host platform need to be executed. This gives virtual machines using purely interpreted bytecode a significant speed disadvantage over binaries compiled to the native architecture of the host. Just-in-Time compilation is the way to go when it comes to performant, bytecode oriented virtual machines that can compete with traditional native binaries. But developing a JIT is hard and one would have to be developed individually for every virtual machine architecture and every implementation of a concrete virtual machine.

This is where VMKit [GTJ$^+$10] comes into play. It provides a framework to develop virtual machines that aims to ease tasks that are common in the implementation of many virtual machines off the shoulders of the programmer. By using VMKit the amount of code that is needed to implement a competitive virtual machine is reduced by an order of magnitude, because the framework already provides implementations for threads, garbage collection and many other functions needed by nearly all virtual machines. One of the tools it uses to reach this goal is the rapidly advancing Low Level Virtual Machine (LLVM) [Lat02]. This compiler framework includes a powerful JIT [LBG02] that can perform binary translation on bytecode written for the Low Level Virtual Machine.

Because it is easier to write programs with Java or C# than in more low-level languages, it would be advisable to have virtual machines available for these languages in the L4 Runtime Environment (L4Re) [l4r11]. By not only providing the virtual machines, but also bindings to the features of the L4 core such as capabilities and Inter-process communication (IPC), it would be possible to develop L4 applications in Java and C#. As porting individual virtual machines for those languages is a difficult task the approach of porting VMKit seems to be much more promising, and would also provide for faster porting or implementation of other virtual machines in the future.

In this document I want to describe a general approach to compile contributing applications for the L4Re platform, with minimal modifications to the applications and their build system. This approach is also taken to port the VMKit components and details on problems I encountered during the porting of them will also be given. In the implementation chapter I present different ideas for interfacing virtual machines with L4Re features and detail the implementation chosen with an example in the Java programming language that is available on top of VMKit. Because the implementation given is only an example that shows the feasibility of implementing L4 services in Java, its shortcomings, as well as a proposal of a better interface, will be discussed. The evaluation chapter contains details on measurements I did to evaluate the performance of L4 clients and servers implemented in Java. As a conclusion of this thesis, a suggestion for further work will be provided as well as details on the current deficiencies and bugs of the ported VMKit/J3 version.

2 Background and Related Work

Because the goal is to port VMKit and J3, which is the Java virtual machine implemented on top of it, to L4Re I first want to give a short introduction to VMKit itself. As VMKit has several dependencies, namely a POSIX environment and the LLVM Just-in-Time compiler, as well as a Java classpath, those will be discussed afterward. Because L4Re is the target that VMKit will be ported to I will also give a short introduction to the L4 Runtime Environment.

Finally, I give a description of existing virtual machine implementations using the LLVM framework and of existing and previous virtual machines implemented on or ported to L4.

2.1 VMKit/J3

As managed runtime environments become more common in all areas from web servers [jav11] to entire operating systems [HL04] there is also an increasing need for research on topics such as runtime speed, verification, garbage collection and many more. The problem is that currently it is a daunting task to implement a whole managed runtime environment for the sake of research. Many of the tasks needed to do so are monotonous and often with no direct relation to the topic at hand. This is where VMKit comes into play. It provides a *substrate* [GTJ$^+$10] for implementing managed runtime environments easing many of the more mundane tasks like developing a JIT or a garbage collector that are often not even related to the researcher's or developer's topic of interest. Doing this, whole managed runtime environments can be implemented in less times and using less amount of code. By using the facilities of VMKit the original authors were able to deliver an implementation of the .NET runtime CLR within a month, using significantly less lines of code than any other implementation [GTJ$^+$10].

The inner workings of VMKit and its interaction with LLVM and other tools are described in detail in the 2010 paper by Geoffray et. al. [GTJ$^+$10], which is the basis for the current implementation of VMKit. The goal of VMKit is not to provide an extremely fast and optimized virtual machine for any already existing VM specification but rather to ease the development and research for future virtual machine implementations and new concepts for virtual machines.

To ease the implementation of individual VMs, VMKit splits the VM in two parts, as illustrated in figure 2.1. One part is the so called *substrate* provided by VMKit itself and the other is the so called machine runtime environment (MRE). The substrate provides the essential features of most virtual machines and is implemented in such a generic fashion that it is adaptable to different virtual machine implementations and does not impose any object model or other method dispatch strategies [GTJ$^+$10]. Garbage collectors, thread management and the Just-in-Time compiler are implemented in the substrate, because these components are needed by virtually all VMs in existence. However, it just provides

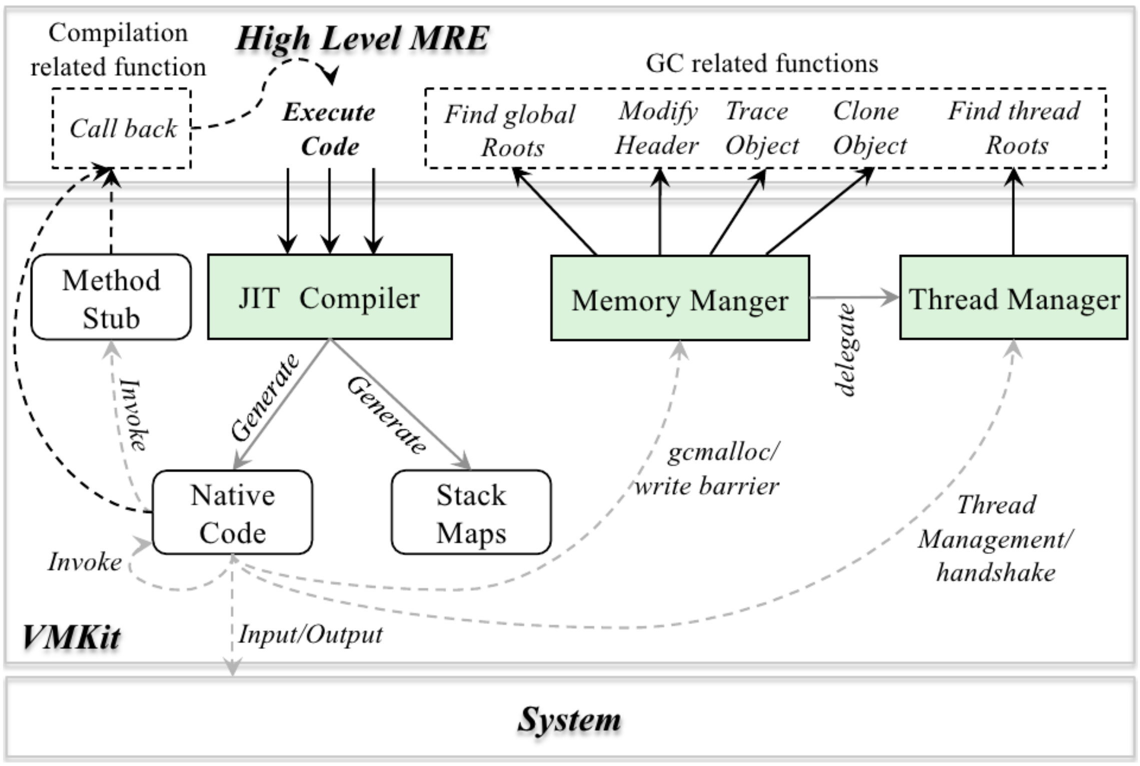

Figure 2.1: Interaction of MRE and VMKit. Dashed boxes are to be implemented in the MRE, solid black errors represent calls between MRE and VMKit. The grey arraows represent calls inside VMKit. The call back is used for JIT compilation, when VMKit needs to signal that new code needs to be generated [GTJ+10]

them and does not force the virtual machine implementation to use them. This way developers are free to, for example, implement a non-garbage-collected MRE or one that does not use threading [GTJ+10].

Two virtual machines have been implemented on top of VMKit so far. Currently only a Java VM named J3 is maintained. There was also a CLR implementation for the .NET virtual machine in previous versions, but it has been abandoned. Because of this the following chapters will only discuss the J3 VM, but are, apart from Java specifics, also applicable to other virtual machines implemented on top of VMKit.

The following components are used by VMKit and the J3 VM: [GTJ+10]

1. A standard C++ library
2. The LLVM JIT for Just-in-Time compilation and the LLVM API for IR generation
3. MMTk or boehm-gc for garbage collection
4. The POSIX thread library for threading support
5. zlib for jar file support
6. The GNU Classpath as a Java classpath implementation

Three of those six components, namely the zlib, standard C++ library and the POSIX threads library pthreads, are available in the L4 Runtime Environment (L4Re). Because of this I first want to give a short introduction to L4Re.

2.2 L4Re

The L4 Runtime Environment (L4Re) [l4r11] is the basis for application development on the L4 based Fiasco.OC [LW09] microkernel, and as such also the basic framework for porting VMKit. L4 was originally developed by Jochen Liedtke in assembly language [Lie95]. Its kernel interface was implemented in C++ under the name Fiasco by the operating system group at TU Dresden [fia11]. As Fiasco is a microkernel, it provides only the most basic features needed to access the hardware and delegates work to user space wherever possible. Usually application developers expect the operating system to provide basic functionality such as threads, a standard C and C++ library, support for shared libraries or a networking subsystem. Those are provided by L4Re, which is the current user space implementation used by the group in Dresden. It aims to be POSIX compatible. POSIX is a standard describing an interface application against which developers. It requires for example to provide a threading library called pthreads and a C runtime. The C runtime used in L4Re is the uclibc and as a C++ runtime the official GNU libstdc++ is used.

The L4Re system and the Fiasco.OC core are capability-based and provide support for fast IPC between tasks. Fast IPC is crucial for competitive performance in microkernels, because IPC is the primary communication mechanism between tasks and also for syscalls. These features of the L4 system are to be integrated into VMKit or the J3 virtual machine to make them accessible from within the languages implemented by the VM and to give developers the possibility to use these L4 mechanisms in a high level language context for faster application development. When doing this, one evaluation point should be the IPC performance, because implementations that do not provide high performance IPC are likely too slow to be considered practical in a microkernel environment.

VMKit does also provide threading support to the MREs implemented on top of it. Because many managed runtime environments support some kind of thread model this is considered crucial. One of the most widely used thread libraries is the pthreads implementation of the POSIX standard for threads [oEEE95]. This implementation is also available on L4Re. The features of pthreads that are used by VMKit are mainly thread creation and destruction as well as communication between threads for the purpose of synchronization during garbage collection.

As a standard C++ library a package named libstdc++v3 is available on L4Re. Despite its name it does also provide a standard C++ library for current versions of the Gnu Compiler Collection (GCC). It is necessary to build any C++ based programs and provides standard objects like iostreams and the std namespace.

The Java VM J3 does also require a library to read and extract jar files, which are only zip files with an additional header. To do this it uses the zlib library [zli11] that is also available as a L4Re package in the development svn branch.

2.3 LLVM

The Low Level Virtual Machine, which is used by VMKit, is not so much a compiler but rather a collection of technologies used in building compilers and toolchains [llv11b]. It was first presented in the Master's Thesis of Chris Lattner in 2002 [Lat02] and much of its inner workings are also detailed in the 2004 paper [LA04] by Chris Lattner and Vikram Adve. As mentioned in the later paper's title, LLVM aims to provide lifelong program analysis and, especially, optimization. This encompasses transformations of the code at compile time, link time, during the deployment of the application on the target platform and during the execution, as well as between executions by the use of profiles taken while it is running [LA04]. The compile time optimizations are mainly language specific ones, performed by the front end before the code is translated to the LLVM bytecode. This intermediate bytecode is one of the main features of LLVM and is used to represent the code on which LLVM works. The bytecode uses an abstract RISC-like register machine [LA04] with an unlimited number of registers, which are used in a typed fashion, and also defines an intermediate assembly language for programming this virtual machine directly. This language is called the *intermediate representation*. LLVM does, however, also provide an API that can generate such code to make implementation for front-end designers and for others wishing to translate code to LLVM bytecode easier.

The bytecode, as well as the intermediate code, are designed to be most suitable for program analysis and optimization by the compiler, providing contextual information that are vital for the components of LLVM to make informed decisions on how to optimize and analyze the program. This is also where the type information come into play. It is not used to provide type safety as commonly found in languages like Java or C#, but to validate complex transformations on the code, preserving its meaning while altering its instruction forms. Those transformations are then applied in later stages of the compile process. Interprocedural optimizations can be performed on the code by both the compiler front end and the LLVM linker stage. The runtime and deployment optimizations are achieved by instrumenting the code and identifying sections that are often executed or run most of the time. Those can then be dynamically analyzed and optimized by the runtime optimizer and the optimized code, which is directly generated, can be inserted on the fly by modifying the jumps to it [LA04]. Those aggressive transformations and optimizations can only be performed, because the type information, which are carried with the LLVM bytecode, provide sufficient information on the types of individual data.

Besides the LLVM bytecode and intermediate representation the Just-in-Time translator is of particular interest for this work, because VMKit incorporates its components heavily and uses this part of LLVM to generate the code that is then executed when a function in Java is called. Only by the use of a JIT translator, programs can be executed with the performance needed to compete against native applications, at least as long as the code and structural checking features of the language should be preserved. If these features are not of interest, then also an Ahead-of-Time compiler could be used, which would compile the code directly to a native binary. Such an AOT Compile is also provided by VMKit through the usage of LLVM bytecode and its bytecode compiler. The Just-in-Time binary translator uses a lazy approach compiling functions only as they are called. This reduces the startup time greatly from approaches where all code is translated to the native platform during the loading of the program, which significant delays in application startup.

2.4 Garbage collection

Because many high level programming languages that run on virtual machines, provide garbage collection to free no longer used memory, VMKit does offer several garbage collector implementations hat can be used in MREs. They all use a generic interface also seen in figure 2.1 on page 4 and because of this it is possible to exchange the garbage collectors without modifying the MREs that use them. Currently the boehm-gc [jBS] and the MMTk [Bla04] garbage collector toolkit are provided as external tools for garbage collection. Beside these two, there are also lightweight garbage collectors called *single-map* and *multi-map* available. These garbage collector sare directly part of VMKit and are precise garbage collectors but are also slow.

The usage of boehm-gc or MMTk was discarded for several reasons. According to a talk by Nicolas Geoffray [Geo09] boehm-gc is a slow garbage collector because it is conservative. This means it scans memory and interpretes any values as addresses. This might lead to false positives, which means memory might not be freed although the object occupying it is no longer in use. Boehm-gc does also not allow for dynamic optimizations of its garbage collection code. MMTk on the other hand is fast and it is easy to implement new strategies for garbage collection. The improved speed is caused by the usage of precies garbage collection by the MMTk garbage collectors. Precise garbage collection means they can find all pointers to other objects in an object and thus do a complete build of the tree of objects that are in use, leading to no false positives.

But there is one drawback. Because MMTk is implemented in Java, it needs to be compiled ahead of time. While this can be done by using the AOT compiler of VMKit, it needs llvm-gcc to be incorporated into VMKit. llvm-gcc is a heavily patched gcc that can generate LLVM bytecode from C files, but has been marked as deprecated by the LLVM project. It will be replaced by dragonegg in the foreseeable future. Because the requirement of such an exotic compiler on the side of the user and the porting of a deprecated software was not deemed to be the best thing to spend time on, the decision was made not to use either of them but rather stay with the integrated multi-map garbage collector for the time being. Although multi-map is an extremely simple and slow garbage collector, it is precise and still provides acceptable performance. It is advisable to reconsider porting MMTk once dragonegg is mature, has a larger install base and is supported by a current VMKit build.

2.5 The GNU Classpath

To run, Java programs expect a number of standard packages to be present. These packages contain the basic functions most applications need and can be compared to the standard libraries or runtime libraries in traditional programming languages like C or Visual Basic. The J3 virtual machine does also provide applications with such a classpath and uses an external package for this. The classpath that was chosen by the VMKit author is the GNU Classpath because it is a well documented and open source implementation of the core classes of the Java 1.5 Standard Edition. However, a few of the more seldom used classes are not fully implemented. An overview of the implementation status can be found on the homepage of the GNU Classpath [gnu11]. There is also extensive documentation available that documents the integration of the GNU classpath libraries into virtual machine implementations [gnu11]. Because the Java code must also interface with the operating system, functions such as access to the operating system's input and output streams or to the filesystem and devices have to be implemented in native shared libraries. Other functions may also be implemented natively for performance reasons. Those native functions must then be called from the Java virtual machine. The method to do call them is via the Java Native Interface (JNI) that is also implemented in the J3 VM. [Lia99] Its libraries need to be compiled for the target platform and expect a POSIX environment.

2.6 Component interaction

The components listed so far are not independent of each other. The most prominent example is the garbage collector that needs to know about the layout of function frames and can only get this information from the JIT to provide multi-threaded garbage collection. However, these components were not originally designed to work together and because of this, VMKit provides a glue between them so that they can interface with each other [GTJ+10]. Using this method the projects must not be modified themselves but can still work together.

Apart from the glue, MREs are required to implement functions that are needed by VMKit, as is also shown in figure 2.1 on page 4. One of the most important functions is the call-back function that is able to generate LLVM bytecode for a given function [GTJ+10]. The usage of such a call back function allows for lazy compilation of the bytecode. As lazy compilation does only compile a function when it is first used, and not upon startup of the application, it does significantly speed up the startup process. Only when a not-yet-compiled function is called this function is passed to the call back function of the MRE, which is then responsible to create the LLVM bytecode that is then passed on to the JIT by VMKit. The call back function can do this translation either by manual translation into LLVM IR or by using the API provided by LLVM to generate the bytecode directly. The code generated this way is then fed to the JIT and executed.

Another set of required functions are the ones related to garbage collection. They provide means to

1. find the root objects
2. trace or clone objects
3. modify the object header

to the garbage collectors [GTJ+10]. The first point is important to find the roots of the trees of objects. Starting from these roots other objects that are pointed to by objects in these trees, have to be found. The trees are constructed using a trace function that has to have knowledge of the structure of objects, and because VMKit does not enforce any specific structure this has to be provided by the MRE. The function to modify the header of an object also has to have knowledge of its structure and is used to, for example, mark the objects in a mark and sweep collector.

2.7 Related Work

Because the LLVM JIT is quite handy when it comes to integration into other projects it is already used by some as a fast way to get a Just-in-Time compiler running. Two of them will be discussed below. I will give a brief description of two examples of them. There are also several languages that make use of virtual machines to execute that have been ported to or directly implemented on L4Re or the previous L4Env.

2.7.1 Other VMs using the LLVM JIT

There already are virtual machines implemented using the LLVM JIT [llv11a]. However, all had to implement their own threading, garbage collection and basic virtual machine building blocks if they wanted to use them. Following are two examples that are also listed on the LLVM website, which use LLVM to differing degrees to implement their virtual machine.

1. An ActionScript 3.0 Just-in-Time compiler that uses the LLVM JIT for native code generation exists. LLVM Bytecode is generated using the LLVM API and then fed to the JIT [Pig08].
2. A virtual machine for an experimental object oriented language called 3C is using the LLVM API to generate bytecode and the LLVM JIT to translate this bytecode to native code [Bar09] .

The authors of the 3C paper [Bar09] encountered a major problem when benchmarking the 3C virtual machine. They quickly ran out of memory for larger benchmarking problems that used large amounts of memory, such as the Fibonacci test with larger numbers. This was because the 3C language has no provisions for either garbage collection or stack frame cleaning. Here an approach to implement the original 3C language on a platform like VMKit might have improved the development speed and lead to an easy implementation of a garbage collector. This would have prevented the problems found in the paper.

2.7.2 Other virtual machines and interpreted languages on L4

There are already several other virtual machines, or languages that could be compiled to them, running on L4Re, and others have been ported to the previous user-land instance L4Env, but are no longer available for L4Re. One of these examples is the Kaffe VM ported to L4Env by Alexander Böttcher in his 2004 Beleg thesis [kaf04]. It has, however, not been ported over to the new L4Re user-land, when the switch was made. But it still provided a fully functional Java implementation in L4Env. Still, this implementtion is different from the port of J3 I performed in that here, the main goal was not the porting of the J3 VM. My main goal was to port the VMKit framework so that future virtual machines could be implemented more easily. The J3 VM can rather be considered a demonstration of the functionality of the VMKit port.

There is also a python port to L4Env done by Aaron Pohle in 2008 [pyt08]. He ported python for use as a shell to L4Env. Further there is an implementation of a Lua interpreter in the form of ned, which is used as a loader for L4Re. All these languages are interpreted or compiled to bytecode and for efficient implementation for all of them a JIT compiler would be beneficial. Python and Lua are not using JIT compilation in their default setup, but it can be provided by external tools that have, so far, not been ported to L4Re. Kaffe can operate either in a JIT or interpreted mode. While all these tools aim to provide support for single languages or a specific bytecode VMKit goes farther than that. It inserts a further layer in between the operating system and the virtual machine, which makes implementing new programming languages or technologies for virtual machines easier. It aims to provide for a fast implementation of JIT, thread, and garbage collection functionality. As such it might even help for future ports of other languages to L4Re, by making integration of a JIT easier and thus the resulting virtual machine faster.

3 Porting

When porting applications to the L4Re system there are several possibilities on how to do this. In this chapter I will first describe four goals I set and the reasons for these goals. Then I will discuss the possibilities for porting and how they match the goals set before. I will then describe the generic approach I chose in more detail. Afterward I will apply this approach to the GNU Classpath, LLVM and VMKit itself to compile them for L4Re and describe problems I encountered while doing so. Their solution might also help other porting efforts in the future.

3.1 Goals

To evaluate the different possibilities to compile existing applications for L4Re I first want to describe four goals against which I want to measure the methods methods I discuss.

1. Reusability for other applicationility
2. Easy maintainability
3. Only moderate requirements on the users build system
4. Comparable or equal functionality to the Linux version

The first point describes the main goal, which is to not only devise a way to port this one program, but to provide a method that can be applied to a multitude of programs that could be ported to L4Re.

While the first point will make porting other applications easier, the second will improve the adoption of newer versions of the same package. Modifications should, if possible, not be made to the source code of the contributing package, but rather to the L4Re implementations of the libraries that have conflicting implementations. In the best case no modifications of to the contributing package's structure and files will be necessary, except to inform its build system of L4Re as a new build target. This will make it easier to update packages and, because of this, will guarantee a low amount of maintenance necessary for them and will also help with the adoption of new versions. New features of the packages, in my example of VMKit and LLVM, will be available faster to users of L4Re this way. Should features be needed by the contributing package that are currently not available in L4Re they should not be worked around in the program's code but rather be made available in the L4 Runtime Environment. This will also benefit other programs that might need these features and ease future porting efforts. If making required functionality available on L4Re is not easily possible, the features should be disabled during the configuration of the package.

The third goal aims to make compiling the package as painless as possible. The goal is to keep the requirements on the user's build environment at an acceptable level and to require, if possible, only the standard development toolchains already available on the user's system, and no exotic compilers or other nonstandard tools. While it is not always possible to achive this goal, the requirements should always be kept in mind when adapting external contributing packages.

The final point is that the functionality should be kept as close to that of the original Linux version as possible. While this point, most of the time, should not be prioritized over the others it is an important part of the porting process and should be weighted against them. For example it is important for the J3 port that all Java classes supported by the Linux implementation do also execute the same way under L4Re.

In general fulfilling these goals is a matter of weighing them against each other.

3.2 Possible approaches

When porting an application to L4Re, there are two possibilities. One is to just put the application into a new package and provide the build system of L4 with the C files to be compiled and linked together. These files must be explicitly named in the programs Makefile. The other is to write a custom Makefile for building contributing packages in general. This Makefile calls the configuration and building stages of the contributing package's build system. Using this second option, the package just needs to be dropped in place - thats what I call the *drop-in* solution. The first approach is only suitable for small applications with a limited number of source files and a simple build system, because most larger software projects and, in my example LLVM and VMKit, require sophisticated build systems that feature many configuration options and definitions as well as multiple binaries that need to be built. The approach with the a special Makefile naming all the C files does not work for them.

It would be possible, but would mean a great amount of unnecessary work and also that future versions of them could not just be *dropped in*. Such a system does not satisfy my first goal, because one would have to write a highly specific Makefile for each application and application version that is to be ported. This highly specific solution is also in contrast to the second goal of easy maintainability. The Makefile would be too specific to a single version, relying on directory structure and filenames of the individual contributing package.

The second possibility is also complex, but I would only have to do most of the work just once. One could compile the package using its native build system but would have to make it aware of the new L4Re architecture, so that it detects that it has to cross compile. As L4Re in general can be considered as POSIX-like, it should be possible to just compile the applications for this platform by providing the build and configuration system with a new architecture. This architecture should compile with the same options as the corresponding Linux architecture. To use this option, the build system needs to have the possibility to cross-compile for a different target. It does so by accepting corresponding options during the call of *configure*.

For this to work, a cross-compiler must be provided. Because the platform GCC is sufficient to generate code for L4, there only needs to be a wrapper around it that automatically adds options such as *-nostdlib* and information about the L4 binary layout in the form of a linker map file. It also needs to provide the platform GCC with the necessary libraries that need to be linked against to resolve all symbols successfully. Luckily, such a wrapper already exists in the L4Re development environment and just needs to be provided with the necessary information such as the required libs. It is called *l4re-gcc* and automatically passes the correct flags on to the hosts GCC, so that it can create binaries for L4. l4re-gcc can be passed to the configure script as the compiler using the CC option. Some build systems that use C++ source code do also need to call g++ and need a cross-compiling g++ to generate binaries for L4Re. l4re-gcc does already handle g++ calls, but it needs a flag to do so. This leads to the problem that l4re-gcc cannot be used as a C++ *and* C compiler at the same time. I worked around this problem by implementing another wrapper for l4re-gcc named *l4re-g++* that only passes its arguments to l4re-gcc and sets the flag to compile C++ code. It just contains the following one-liner:

```
#/bin/sh L4RE_GCC_CXX=1 $(dirname $0)/l4re-gcc $@
```

The L4RE_GCC_CXX flag is used to indicate a C++ build to l4re-gcc. This wrapper can then be passed to configure scripts as C++ compiler using the CXX option, whereas the l4re-gcc can still be passed as C compiler using the CC option of the configure script.

3.3 Implementation of the Makefile

For maximum reusability I decided to implement the make system by using a Makefile that the user does not need to modify and a file called Makefile.cnf. The latter file defines a number of variables and - if desired - targets,that are used to configure the behavior of the Makefile. Because most packages will need patching, a support directory is defined in the contributing packages structure. This contains patch files and files that need to be applied in different stages of the build process. They are selected based on the prefix of their filename. The following prefixes are used by the Makefile:

- **l4re_unpack_**
  These patches are applied immediately after unpacking the downloaded contributing package
- **l4re_reconf_**
  These patches are applied before a reconfiguration of the package is initiated and can be used to make the L4Re target known to the package's build system. A reconfiguration is only performed, when at least one patch with this prefix is found.
- **l4re_postreconf_**
  These patches are applied after a reconfiguration of the package has been performed. They can be used to correct mistakes by the autotools but should not be needed in general.
- **l4re_postconf_**
  These patches are applied after the configure command of the package has been called. This feature is necessary if configure misdetects the features of the host system due to the cross-compilation environment.

The patches may be provided either as .patch files that are fed into the patch command or as .sh files that are executed in the source directory. Patches and scripts are executed in the source directory except for the postconf scripts and patches, which are executed in the build directory.

The following variables, which can be defined in Makefile.cnf, are used by the Makefile for now:

- **L4_REQUIRES_LIBS**
  In this variable a list of all the libraries that are required to build the package should be provided.
- **CONTRIB_VERSION**
  The variable is used only in the configure file to make it easier to adapt for a new version by changing just one parameter in the Makefile.conf File.
- **CONTRIB_URL**
  This variable is used as the path from which to fetch the package. It must be parsable by wget as an URL.
- **CONTRIB_RECONF**
  This variable contains the path to the reconfiguration script, usually called AutoReconf.sh relative to the contributing package's root. It is used to start the reconfiguration when patches are applied to the source files of the autotools toolbox. It is only used if l4re_reconf patches are found.
- **CONTRIB_SUBDIR**
  If this variable is set to any value then the binaries and libraries of the package will be installed into their own subdirectory in the L4 build tree, which is created by the Makefile. The name of the subdirectory is the value of this variable.
- **CONTRIB_CONFIGURE**
  All switches that must be passed to the configure script to enable or disable individual features should be provided in this variable. The –host, –build and –target options should *not* be given, because they are already set by the Makefile.
- **CONTRIB_ACLOCAL**
  Sets the version of aclocal that should be provided in the alias function to the AutoReconf.sh script.
- **CONTRIB_AUTOCONF**
  Sets the version of autoconf that should be provided in the alias function to the AutoReconf.sh script.
- **CONTRIB_LIBS**
  All files that are listed in this space-separated variable will be symlinked to the libs directory of L4 if they exist. The path is relative to the build directory.

- **CONTRIB_BINS**
  All files that are listed in this space-separated variable will be symlinked to the bin directory of L4 if they exist. The path is relative to the build directory.

These variables are evaluated by the Makefile. The packages are always built out of tree. The Makefile.cnf may also specify additional targets with names similar to the prefixes of the support files. For example the target for after the unpack phase is called l4re_unpack::. These targets are executed after the corresponding support files have been run.

All these measures should cover the most common cases that are needed when compiling external packages for L4Re.

3.4 GNU Classpath

The GNU Classpath's build system is simple enough, so that it could also be easily transferred to a regular L4 package with its source files listed in the makefile. It could then be directly built using BID. For two reasons I chose not to do it this way. First and foremost, the Classpath was a tryout for my methodology of compiling using the *l4re-gcc* wrapper and the new Makefile system. Its simple structure made it easier for me to debug and analyze the build process, to quickly verify the chosen method of simple cross compilation by using the package's own build system. The second reason was that, although writing a Makefile would probably be easier in this case, it would still not meet the second of my goals, as defined previously. It is not future-proof to manually write makefiles that depend on the structure and filenames of a contributing package. When a package changes these structures, it can still be built using its own build system whereas a custom makefile would have to be adapted to the new structure. So the goal of a drop-in solution can be more easily achieved by using the native build system.

The first part of making the Classpath (or for that matter all the components) compile for L4 is to analyze its build system and the options needed when compiling. What I had to modify in the build system was to patch the config.sub file to make the L4Re system known to the configure script. No further patches were needed at that point.

I also had to disable three of the features of the classpath because they were not available on L4Re. Disabling those features was done using the CONTRIB_CONFIGURE option:

- The plugin contained in the package is not needed because it is only relevant for web browsers.
- The GTK Peer should not be compiled, because GTK is not available on L4 but it is enabled in the default classpath configuration.
- The GConf Peer should be disabled because GConf is not available on L4.

It turned out that the configure script misdetects three features. It assumes that the files magic.h and epoll.h are available on the L4Re environment while they are not. I patched the created include/config.h header file to solve the problem. Its origin, however, was not clear to me but it seems to be rooted in an errorneous configure script. Another misdetected feature was IPv6 support. The structure sockaddr_in6, which is used to

Listing 3.1: The Makefile.cnf for the GNU Classpath

```
CONTRIB_VERSION=0.97.2
CONTRIB_URL=ftp://ftp.gnu.org/gnu/classpath/\
  classpath-$(CONTRIB_VERSION).tar.gz
CONTRIB_SUBDIR=java
CONTRIB_CONFIGURE=--disable-plugin --disable-gtk-peer --disable-gconf-peer
CONTRIB_LIBS=*.so *lib/glibj.zip
```

detect IPv6 support, is always included in the header files, independent of the actual features selected in uclibc. Ignoring this leads to a compilation error, because the constants used in the IPv6 aware Classpath are not defined unless IPv6 is enabled in the uclibc. Patching the config.h solved this problem as well. The rest of the compilation ran like on any Linux system. The patches were put in the support directory and were applied by the Makefile.

Another problem I found was with the shared libraries. They were not built at first, because the configure script did assume that no shared libraries were available on L4Re. This was not possible to rectify in the *configure.ac* source script for the *configure* file. Because of this I created a patch that modified the relevant parts in the *configure* script, so that the L4Re version behaves the same as the Linux version.

I was not able to test the compiled and installed libraries at this point. They cannot be tested on their own because they are designed to be used by virtual machines. The compiled libraries are installed (linked) to a *java* subdirectory in the libs/ directory. The corresponding Makefile.cnf is shown in listing 3.1. Most of the work is done by the actual Makefile in the *src* directory, which is universal for all contributing packages.

3.5 LLVM

LLVM is the only other component besides the Classpath that is directly needed to compile and run VMKit with the J3 virtual machine. While this piece of software is much more complex than the Classpath, porting it proved to be easy. This was possible thanks to a clean and well documented build system and a clear separation of code dependent on OS functionality and code that just uses these functionalities to provide the functions of LLVM such as the JIT or the assembler.

The build system again uses the automake tools but this time does add sections to the configure script depending on the operating system. Because of this patching the *configure.ac* file that is used to generate the configure script, was necessary and with this also the regeneration of the configure script. The example Makefile.cnf, which performs this patching of the configure.ac, is displayed in listing 3.2. The main problem I encountered, and that first appeared in this package, is that those automake scripts are highly dependent on the version of the automake tools. Even different minor versions may not parse scripts that another minor version does or generate wrong configure scripts that cause strange configuration and build errors to appear. This was the first time I had to deal with a conflict between goals two and three. On the one hand I can require the user to have all possible versions of the autotools available on his System. This would clearly

Listing 3.2: The Makefile.cnf for LLVM

```
 export L4RE_REQUIRES_LIBS=stdlibs libstdc++ libc_support_misc \
     libc_be_sig libc_be_socket_noop l4re_c \
     l4re_c-util libpthread libdl ldso
 export L4RE_MULTITHREADED=y
 export L4RE_LINK_OPTS=$(OBJ_BASE)/lib/$(BUILD_ARCH)_$(CPU)/l4f/libuc_c.a
CONTRIB_VERSION=2.8
CONTRIB_URL=http://llvm.org/releases/$(CONTRIB_VERSION)/\
   llvm-$(CONTRIB_VERSION).tgz
CONTRIB_SUBDIR=llvm
CONTRIB_CONFIGURE=--disable-shared --disable-pic
CONTRIB_RECONF=autoconf/AutoRegen.sh
CONTRIB_ACLOCAL=1.9
CONTRIB_BINS='*Release/bin/*'

l4re_clean::
        rm -rf ../build_native/* ../build_native/.configured;

l4re_unpack::
        cd ../build_native; \
        [ ! -e .configured ] && \i
        ../src/llvm-$(CONTRIB_VERSION)/configure $(CONTRIB_CONFIGURE) && \
                touch .configured; \
        make;
```

violate point three of my goals. Or I could patch the configuration myself, which would violate point two, as this patch would have to be updated for every new version of the package. I decided for requiring the correct autotool for the configuration and only patching the *configure.ac* file. The checking for the correct autotools version is done using a script provided with LLVM that checks for the correct version of those tools and performs the right steps for generation of the configure system in the correct order. Such a file is included with most packages.

The dependencies of LLVM-2.8 are autoconf 2.60, aclocal 1.9.6 and libtool 1.5.22. While aclocal and libtool were available on the build system, the version 2.60 of autoconf was not. This was due to the fact that autoconf had already moved to a newer version number of 2.65 and the exact version 2.60 was needed. Trial and error confirmed that the configure script could also be built with the version available. Because of this, a patch was included, which enables reconfiguration to succeed with any version starting with 2.6.

The conflict, between the different goals was that the autotools were required to be of a specific version, and must not only match in their major version. This version dependency is a big problem because it conflicts directly with point three of my goals. Requiring the user to install possibly deprecated software that might even be no longer available on his system presents a major hurdle. In this case it was possible to work around this problem by patching the AutoRegen.sh script. I included this script in the patch directory and it was applied before the reconfiguration of the application. This worked for the Gentoo system I used to work on this thesis, but would not work for Debian because it does not support a slotted installation of multiple libtool versions side by side. For Debian, this can only be solved by manually installing the correct libtool version.

For the other tools, the Gentoo system I used for building provided an approach to install several versions of autoconf tools side by side. To select a specific version that should be called when aclocal is called the environment variable WANT_AUTOMAKE needs to be set to the desired version or slot. For LLVM this was 1.9. For other operating systems or Linux flavors the approach taken to accept different automake tools might vary. Because of this I included an option to set the aclocal version by adding the CONTRIB_-ACLOCAL variable to the Makefile. I implemented this in a way that the Makefile creates a function called aclocal, which directly calls the specified aclocal version. If no version is specified the generic *aclocal* binary is executed. The usage of CONTRIB_ACLOCAL can also be seen in listing 3.2.

As with the GNU Classpath, header files were again misdetected. The config.h file generated by the configure script needed to be patched because it included its own definition for error_t although this was already present in errno.h. However the configure script only searched argz.h that did not include this definition. It assumed that errno_h was not present although it was. So besides the removal of the wrong error_t definition the config.h also needed to be patched to define the HAVE_ERRNO_H and HAVE_ERROR_T preprocessor constants. I created the relevant patches and included them in the support directory. They are then automatically applied by the Makefile after the configure phase.

As you can see in listing 3.2 the binaries are also installed in their own LLVM directory. For this the option CONTRIB_SUBDIR is set to the name of this directory.

It is important to note at this point, that by merely porting LLVM and its tools, we do not achieve the goal of having a virtual machine akin to JVM or CLR for the L4Re platform, because its intermediate representation is too low level to represent the language features of Java or .NET languages. [LA04]. It designed this way on purpose, to be independent of the programming and, if present, of the object model of the languages compiled to it. It also has no runtime of its own, like the Classpath for JVM or the .NET framework for CLR but instead it is able to define external symbols that must be resolved to system or library functions during or before execution.

3.6 VMKit and J3

The VMKit build system was a bit more difficult to analyze. While they also use automake for their configuration system, they do use Makefiles that are integrated tightly into the build system of LLVM. Not only do they include and link against files from LLVM, but they also use the LLVM tools to compile the files that are written directly in the LLVM assembler language. The usage of the LLVM tools on the platform the VMKit is compiled on is also the reason why I needed to build both a version of LLVM for the L4 Runtime Environment and one version for the architecture of the build host, so that the binaries are available during VMKit compilation. This can be seen in the previous Makefile.cnf listing.

Listing 3.3: Original code to determine username

```

tmp = getenv("USERNAME");
if (!tmp)
  tmp = getenv("LOGNAME");
else
  if (!tmp)
    tmp = getenv("NAME");
  else
    if (!tmp)
      tmp = "";
setProperty(vm, prop, "user.
```

Again, I patched the configure.sub and configure.ac files to include the L4Re target system. The template I used in the patch was a copy of the same section for the Linux target.

The configure.ac also contains information on the location of LLVM source and object files. The auto-detection of the AutoRegen script did not find the L4 package because the directory structure was unknown to it. Because of this I also patched the AutoRegen.sh script, so that it could find the LLVM package in the L4 package tree.

The configuration options used in compiling VMKit came straight from the directions on the VMKit homepage **??**.

However, those switches and the patches are still not enough to make the package build successfully. To achieve that, the variables *LLVMAS* and *LLC* need to point to the LLVM assembler and compiler and both of them must be compiled for the build platform as mentioned before. This was achived by just exporting those variables in the Makefile.cnf.

After those changes are made, VMKIT/J3 will compile fine but still refuses to start. At this point one of the main problems of the porting of VMKit appeared. If errors appeared in JIT compiled code, there is no indication of what the corresponding source code is, making debugging and problem finding a nightmare. Usually it is then only possible to debug by extensive printf debugging and modification of the Classpath's Java files and native libraries. One of the two bugs found this way was a classical dangling-else problem in LLVM's source code that just never appeared in any traditional use case. When the system properties are set, the code seen in listing 3.3 was executed to determine the current users username.

The formating was inserted by me and makes the problem clear. The environment variable USERNAME is not set per default on L4 and thus returns NULL on the getenv call. The first *if* part is executed in this case and checks for the environment variable LOGNAME that is not set either. However, the *else* part is then never executed. It is only taken when tmp is *not* NULL after the first getenv. Because of this the "ultimate" username, being the empty string, is never assigned and null is passed on as a system property. This leads to strange behavior later on in the execution. I submitted the following fixed version seen in listing 3.4 to the VMKit mailing list and it was accepted, so that the patch currently included in the VMKit package will not be necessary for VMKit version greater 2.9.

Listing 3.4: Modified code to determine username

```
tmp = getenv("USERNAME");
if (!tmp) tmp = getenv("LOGNAME");
if (!tmp) tmp = getenv("NAME");
if (!tmp) tmp = "";
setProperty(vm, prop, "user.
```

The other bug I encountered was one originally rooted in the build system but not detected properly because of bogus system behavior. The installment of l4re-gcc that was used in the beginning did not include the position independent versions of libraries when linking shared objects. This led to two kinds of problems. First, the relocation tables in the created objects could not be rewritten upon dynamically relocating a function. And second, it lead to a kind of partial loading of the library so that J3 assumed the library would be available even though the relocation entries pointed to the wrong address. The offset relative to the program's start was still taken as the current executable address. Because of this, the system did not return an error and stop execution but instead failed with a segfault as soon as one of the functions of such a shared library was called. Modifications to l4re-gcc, which included the PIC versions of the L4Re system libraries when building shared objects, ensured that this bug would no longer occur.

Another bug would also appear after the program ran for a short time and is related to the so called StackWalker of J3. To determine the current depth of the stack, the StackWalker steps through the different stack frames. It determines the end of its walk as soon as it has reached the end of the stack, which is stored in a variable. The current position of the walker is compared to the end of the stack by checking for equality. However, for an unknown reason the walker goes beyond the end of the stack causing page faults. I worked around this problem by not comparing for equality but by checking whether the next stack frame is *above* the end of the stack. This hackish solution worked for the tried programs but might be a source of errors in the future and should be further examined. Because the stack is shared between frames generated by code produced by the Just-in-Time compiler and "native" stack frames, debugging this could proof to be rather challenging. One possible source of the error could be a wrong assumption about the layout of either the frames generated by JIT code or the "native" frames.

In the beginning of compiling J3 I found another problem. While L4Re provided pthreads support, so far it did not provide the pthread_kill function to send signals to other threads. This was, however, necessary when compiling J3 without cooperative garbage collection. Because cooperative garbage collection would once again need llvm-gcc this was not an option. Adam Lackorzynski implemented the support for pthread_kill.

For reference I included the Makefile.cnf for VMKit/J3 again. It can be seen in listing 3.5.

Listing 3.5: The Makefile.cnf for VMKit

```
export L4RE_REQUIRES_LIBS=stdlibs libstdc++ libc_support_misc \
  libc_be_sig libc_be_socket_noop l4re_c l4re_c-util \
  libpthread zlib libc ldso libdl
export L4RE_MULTITHREADED=y
export L4RE_LINK_OPTS=$(OBJ_BASE)/lib/$(BUILD_ARCH)_$(CPU)/l4f/libuc_c.a
CONTRIB_VERSION=28
CONTRIB_URL=http://llvm.org/viewvc/llvm-project/vmkit/branches/\
  release_0$(CONTRIB_VERSION)/www/releases/\
  vmkit-0.$(CONTRIB_VERSION).tar.bz2?revision=116299
CONTRIB_SUBDIR=vmkit
CONTRIB_CONFIGURE=--with-gnu-classpath-glibj=/rom \
  --with-gnu-classpath-libs=/rom \
  --with-llvmsrc=$(L4DIR_ABS)/pkg/llvm/src/llvm-2.8/ \
  --with-llvmobj=$(L4DIR_ABS)/pkg/llvm/build
CONTRIB_RECONF=autoconf/AutoRegen.sh
CONTRIB_BINS='*Release/bin/*'
CONTRIB_ACLOCAL=1.9
export LLVMAS=$(L4DIR_ABS)/pkg/llvm/build_native/Release/bin/llvm-as
export LLC=$(L4DIR_ABS)/pkg/llvm/build_native/Release/bin/llc
```

4 Implementing the Java - L4Re Interface

In this chapter I want to analyze the possibility to interface the L4 functions for IPC and capabilities with languages implemented on top of virtual machines. At first, I want to discuss whether there is a possibility to implement this in such a way that it can be trivially integrated into new languages by using the abstractions provided by VMKit. I then discuss possible approaches to implement such an interface. After this follows a description of my example implementation of the chosen approach, which can be used in the J3 Java VM.

4.1 Introduction

My first idea was to implement the interface at the level of VMKit. My hope was that this would result in an implementation that could be easily reused in other VMs implemented on top of VMKit. After looking into the details of VMKit, I found that this approach is not possible for a variety of reasons. Mainly, this is not what VMKit is all about. VMKit aims to remove tasks that are generic for virtual machine design from the shoulders of the VM developer. However, the language runtime libraries are not universal to all virtual machines. On the contrary, each virtual machine has its own, independent, runtime libraries, be it the .NET CLR with the CLI or Java with its Classpath. It would not be possible to write a generic library that would lead to a feature being automagically available on all implemented VMs. Doing so would, first and foremost, impose an object model on the virtual machines, because it needs a common interface between native libraries and the virtual machine objects or functions. This common interface is one of the features VMKit explicitly does not want to provide, to enable the user to implement their VMs as independently as possible. It also does not provide a generic way to call out to the underlying operating system. This must be implemented in the VM layer on top of VMKit and is thus out of scope for the VMKit package.

The only way to implement bindings to L4Re is by providing each virtual machine with its own native platform libraries. In the case of Java, these platform libraries are shared objects bound to Java using the Java Native Interface (JNI). But not all is lost. While there is no way of easing the development of these libraries of the implementers shoulders, the task can still be made easier. With some clearly defined assumptions about the virtual machines in mind, one could easily write a shared library that needs only be called from the virtual machine's native bindings and would hide away many of the more difficult tasks. For the VM language I detail the implementation for Java, because this could be verified with the existing J3 Java VM. The implementation for other object based languages should be close to the one presented here.

4.2 Concepts

There are two ways that can be used to implement bindings between an object based VM language and the L4Re interfaces. The first is to try to reimplement as many of the native L4 objects, such as IOStreams or capabilities, directly in Java. This method is, what I call languages specific objects as they have to be reimplemented in every VM for which the bindings should be provided. The other method is by the use of the native L4 objects in the native libraries and, instead of reimplementing them in Java, providing a wrapper that just references these native objects in the native library.

4.2.1 Language specific objects

By using language specific objects you have to port every object that is relevant for IPC and capabilities to Java by reimplementing its methods in Java and mirroring the object's structure. Only methods that are not possible to implement, even with the knowledge of how they work internally, such as writing to the threads user-level thread control block (UTCB), would need to be implemented in native libraries, minimizing the size of these libraries and keeping them as a thin abstraction layer. Such methods are mainly those, which directly manipulate structures like the UTCB or do calls to the Fiasco kernel. While this method might seem desirable at first it bears the problem of redundancy. If one were to implement a second VM on top of VMKit all these libraries would need to be rewritten for the new platform, and every change in the underlying structures would lead to changes in every VM. Taking IPC as an example one could imagine that the Java implementation of the IPC stream only needs to call out to native code whenever it changes the UTCB, because this ultimately is what is abstracted by the iostreams, and for doing the actual call. While this only needs two methods (write to UTCB and call) in the native libraries, it also always leads to changes in the Java code and the code of all other VM's interfaces to L4Re, whenever the implementation of IOStreams or other reimplemented objects changes. Such changes are clearly not desirable and would lead to a hard to maintain library that is bound to not keep pace with changes to the platforms libraries. It would also break the abstraction provided by these objects and require explicit knowledge about their internal behavior.

Because the method with language specific objects has to perform less calls using the native interface, one might assume that language specific objects are faster than the native objects and wrappers described in the next subsection. I will evaluate this possibility in the evaluation chapter.

4.2.2 Native objects and wrappers

It would be much more desirable to only implement a wrapper around the native objects and to only expose the public interfaces to the VM. Taking again the example of IOStreams and Java, this would mean implementing calls to put a byte, word, float, and so on into the IPC stream in a Java IOStream object and each of these calls would map to a native method that performs the action. The actual iostreams would never be implemented in Java but only these interfaces. When the Java IOStream object is created or an IOStream object is received by an IPC call, this object is put into a list in the native library. Only the index into this list is then passed on to the virtual machine. This allows for the iostream object to once again be isolated from the Java iostream object and changes to the internals of it do not need to be propagated to either the Java or the native libraries implementation. They just use the object they get.

While this method of designing the interface is nice, and clean, it also means that a certain amount of native lists need to be managed. There must be one list for each object type that is to be wrapped in the VMs language or, alternatively, one list that contains all object types. I would discourage the latter method because it increases the risk of casting entries to the wrong object type. The management of these objects needs to be carefully worked out to prevent memory leaks in the native libraries. It also requires a destructor in the virtual machine that can call a free function in C, to free elements of the list once the objects in the VM are destroyed. Virtual machines that do not implement such mechanisms *will* cause leakage when using these methods. The only way then is to require the developer to manually call a native function, which frees the structures in the native library, once the object is no longer used.

4.3 Implementation and suggestions for improvement

I will provide an example implementation that I implemented to demonstrate the performance and feasibility of implementing L4 IPC servers and clients in Java. I will then also make a suggestion on how this implementation can be improved and what is missing to provide a complete IPC library for Java.

4.4 Example Implementation

The example implementation consists of a Java interface that servers and clients can be developed against, with classes to inherit from and a native part that is implemented as shared libraries using the JNI. The structure of the package can be seen in figure 4.1.

4.4.0.1 The Java interface

Based on the interfaces seen in figure 4.1, I implemented the two classes ExampleCl and ExampleSvr that closely mimic the behavior of their native counterparts in the client/server example provided with L4Re. They work as intended and, especially, they work together with each other as well as with the native programs. This means either the client or the server or even both can be implemented in Java, enabling IPC between virtual machines and between programs running in virtual machines and running natively.

Listing 4.1 shows a simplified version of the example client implemented by me. I designed the interface in such a way that implementing a client feels similar to doing the same in C++. First, the capability is fetched and then an IOStream object is created. Afterward, data is put into the IOStream that was just generated. However, because Java does not support streams, data is put into the stream by calling a *put* method on the stream's object. Then, the call is made on the stream using the capability that was

Listing 4.1: Code for example Java IPC client

```java
public class ExampleCl {
  private static final int OPCODE_NEG = 1;
  private static final int PROTOCOL_CALC = 0;

  public static int function_neg_call(Cap server, int val) {
    Iostream s = new Iostream();
    s.put(OPCODE_NEG); s.put(val);
    l4_msgtag_t res = s.call(server, PROTOCOL_CALC);
    if (res.has_error() != 0) {
      System.err.println("IPC␣Error");
      return 0;
    }
    return s.getInt();
  }

  public static void main(String[] args) {
    Cap c = L4Re.Env.env().get_cap("calc_server");
    if (!c.is_valid()) return;
    int res = func_neg_call(c,8);
    System.out.println("The␣result␣of␣neg␣8␣is␣"+res);
  }

}
```

Listing 4.2: Code for example Java IPC server

```java
public class ExampleSrr extends Server_object {
  private static final int OPCODE_NEG = 1;
  private static final int L4_EOK = 0;

  public static int dispatch(int ios) {
    Iostream s = new Iostream(ios);
    int opcode = str.getInt();
    switch (opcode) {
      case OPCODE_NEG:
        int val1 = str.getInt();
        str.put(- val1);
        return L4_EOK;
      default:
        System.out.println("Unknown␣Op");
        return -1;
    }
  }

  public ExampleSvr(String capname) { super(capname); }

  public static void main(String[] args) {
    ExampleSrv srv = new ExampleSrv("calc_server");
    srv.loop();
  }
}
```

received in the beginning. The L4 message tag from the communication is kept and checked for errors. The generic L4 IPC methods were put into a IpcMethods object. Similar to writing data to the IOStream, data can be retrieved by a *get* method, but because methods cannot be overloaded based on their return type, the type of the data to be read must be given explicitly in the functions name. Although this suggests that the data in the stream would be of the desired type, it is not. The type is not checked, but the next data is just cast to the type the get function returns.

The example server, which I also ported, is shown in listing 4.2. It is quite compact and features most of the functionality of its C++ counterpart. Things I did not implement, like the check of the protocol, were only left out for reasons of simplicity and not because they could not be implemented. The server has to inherit from the Server_object class that contains most of the underlying implementation and abstracts details like the server loop and the dispatch function that has to communicate with the native library. Similar to the native version the server has to be sent into a server loop to wait for incoming data. Because of this it might also make sense to implement this in a separate Java thread if one wishes to do other things concurrently. I did not do this because being the IPC server is the only job of this example program.

The central part of a server object is the dispatch function. Here the IOStream is created with the ios integer passed to this function. This integer is an index into the iostream list of the native library.

The usage of the IOStream is again similar to its native handling and the opcode the client put into the stream can just be read. The result is calculated based on the read values and written back to the stream. At the end 0 is returned to signal success or -1 to signal failure.

As can be seen with the previous examples, the implementation of a client and server in Java is possible and not much harder to do than the native implementation. However, the current implementation only deals with the things shown in these examples. A version for production use would have to implement all the methods in figure 4.1 and provide access to all methods provided by L4Re for capabilities, IPC and IPC error checking.

4.4.0.2 The native library

The implementation of the native interface caused problems at first. Because the native part of the server is supposed to enter an infinite server loop after setting up communication, it is not suitable to be done in the same thread in which the native library is running. This would lead to blocking of the virtual machines thread that runs the native calls upon the start of the server loop. It is also difficult to create objects or invoke methods on a running virtual machine from a function, such as the dispatch function, which was not directly called from the JNI handler of this machine. It was thus decided to move the server loop to an extra thread and to provide a dispatch method stub in this thread. The figure 4.2 shows the typical IPC communication flow for the framework. The dispatch function blocks on the serversem semaphore to stall the communication until the Java handler does process it. The Java-side dispatch function itself is in an endless loop that always calls the *native_wait_server* function. In this function the semaphore of the native dispatch thread is unlocked and the function then waits for the IPC stream to be passed to it using a global variable. Once it gets the id of the IPC stream it calls the *dispatch* method in the Java code, passing the id on. Once the Java dispatch handler is done, control is returned to the *native_wait_server* method in the native library and the result is signaled to the server threads dispatch function, which then returns the result code.

The implementation of this was more complex than I thought at first, because the use of mutexes overwrites data in the UTCB that also contains the IPC stream's data. This led to the client and the server reading wrong data. But it was also not possible to use busy waiting for the communication. While it would have been possible technically it did not make sense, because the server thread might have to wait for a long time for the Java VM to call the *native_wait_server* method and would burn CPU cycles in the meantime. The solution to this was to create a copy of the input stream before waiting on the semaphore. While this seemed like an ideal solution, we can, of course, not write to an input stream. Because of that I implemented an IOWrapper class, shown in figure 4.3, that can contain, IOStreams and IStreams. There *always* has to be an IOStream but when there is an additional IStream this one is returned upon a call to getI(). This enables reading from the copied stream while writing to the original IOStream and thus to the UTCB.

Another problem I encountered was that when the registry server is created using the default constructor, IPC is always bound to the main thread, instead of the thread in which the constructor is called. This caused IPC to never reach the server thread because it tried to target the main JNI native method thread. A solution to this was to get the thread capability of the server thread and pass it to the registry server. This bound the IPC server to the correct thread and communication worked without a problem.

As mentioned in the beginning the L4 objects are not passed directly to the Java code, but instead are stored in lists. This method requires for the Java VM to free the space in the list upon garbage collection of the corresponding Java wrapper objects. This cleanup routine is not implemented at the moment, leading to memory leakage in larger scale or longer running servers, but can be done by adding a destructor to the dummy objects that frees the native list elements.

4.4.1 A cleaner interface

While my implementation, which I discussed in the previous section, basically works and is a nice demonstration of the general possibility to implement servers and clients in Java, it has several shortcomings:

1. The implementation does not support multiple servers. For example the same application could process the protocol for console and i/o but it is not possible to implement this cleanly with the current interface.

2. The native lists are not transparent to the Java programmer. There are indices into these lists passed at placeis visible to the programmer. Wrapper objects should not have to be created based on these indices by the application developer.

3. The handling of the list indices, the object destruction and similar repeating tasks should not be implemented in each wrapper object. They are redundant and make changes to the interface between native objects and Java wrappers more difficult.

These problems can be rectified. I suggest the class structure that is shown in figure 4.4 as one solution. As can be seen, the first point in the list of shortcomings is addressed by the servers not only implementing the server object, but instead by using a singleton *master server* where individual servers can register the protocol they speak. The master server's dispatch routine then passes on the iostream to these server, based on the protocol found in the message tags.

The lacking abstraction, as mentioned in the second point, is seen in two things. First, there is the problem that to several methods the native object id is passed instead of the dummy object. An example here is the current dispatch implementation. This can lead to server problems including the possibility to create duplicates of the same IOStream object, which would lead to severe problems when the destructor of one of them cleans up the entry in the native list. A solution to this would be to make all the constructors of the dummy items private, add a private flag indicating whether it was created by native methods (*isOriginal*). Objects with this flag set would then only be created from the native library. Copies of these objects would still be possible, and even be encouraged, by providing a *copy* method, which returns a copy of the current dummy object, but with the *isOriginal* flag set to false and an internal variable storing a reference to the original dummy object. This guarantees that the "original“ dummy object will never be cleaned up as long as copies are still in use and the copies can check the *isOriginal* flag in the destructor to decide whether to clean the native list entry or not. Only the object that was originally created by the native JNI method would call the native clean up method.

Because the current suggestions are getting quite complex it also adds to the third point of redundant implementations. Such redundancies are found in the Java part in the form discussed before and can be avoided by providing a base *L4_Wrapper* object, from which all wrapper objects (IPC-Streams, Capabilities, etc) inherit. This dummy object is also show in figure 4.4 The *L4_wrapper* class also takes care of freeing list elements in the native library, abstracting this away from the programmer.

As for the native implementation one could keep most of the current structure, but it would be advisable to move most of the list handling functions to an external library that could be linked against. This limits the redundancy in the implementations of native libraries for the different MREs implemented on top of VMKit.

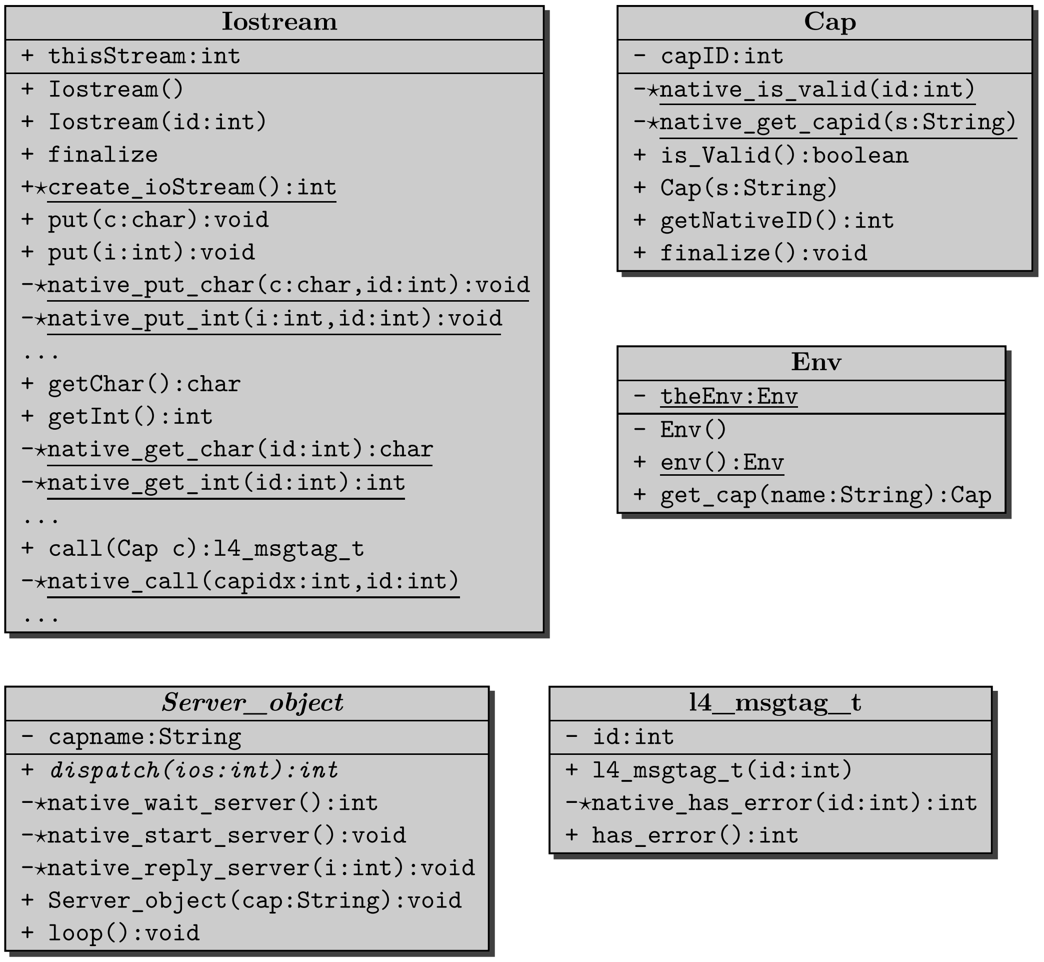
Iostream
+ thisStream:int
+ Iostream()
+ Iostream(id:int)
+ finalize
+⋆create_ioStream():int
+ put(c:char):void
+ put(i:int):void
-⋆native_put_char(c:char,id:int):void
-⋆native_put_int(i:int,id:int):void
...
+ getChar():char
+ getInt():int
-⋆native_get_char(id:int):char
-⋆native_get_int(id:int):int
...
+ call(Cap c):l4_msgtag_t
-⋆native_call(capidx:int,id:int)
...
Cap
- capID:int
-⋆native_is_valid(id:int)
-⋆native_get_capid(s:String)
+ is_Valid():boolean
+ Cap(s:String)
+ getNativeID():int
+ finalize():void
Env
- theEnv:Env
- Env()
+ env():Env
+ get_cap(name:String):Cap
Server_object
- capname:String
+ dispatch(ios:int):int
-⋆native_wait_server():int
-⋆native_start_server():void
-⋆native_reply_server(i:int):void
+ Server_object(cap:String):void
+ loop():void
l4_msgtag_t
- id:int
+ l4_msgtag_t(id:int)
-⋆native_has_error(id:int):int
+ has_error():int

Figure 4.1: Class diagram of the current implementation of the L4Re interface. Methods marked with an asterisk are native methods implemented in the l4native library

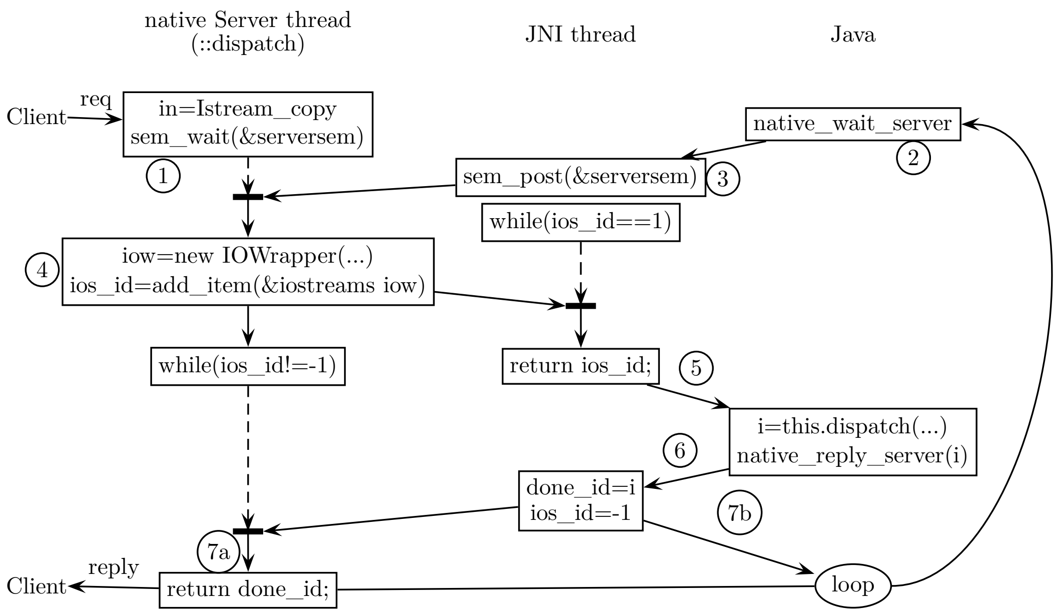

Figure 4.2: Flowchart for the IPC with a L4 server implemented in Java

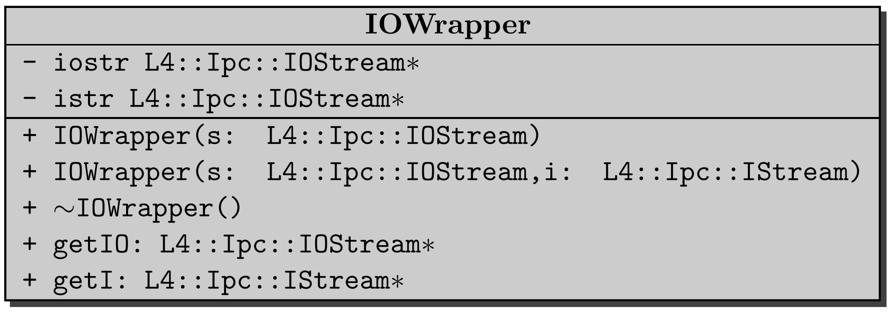
IOWrapper
- iostr L4::Ipc::IOStream*
- istr L4::Ipc::IOStream*
+ IOWrapper(s: L4::Ipc::IOStream)
+ IOWrapper(s: L4::Ipc::IOStream,i: L4::Ipc::IStream)
+ ∼IOWrapper()
+ getIO: L4::Ipc::IOStream*
+ getI: L4::Ipc::IStream*

Figure 4.3: Wrapper for IOStreams

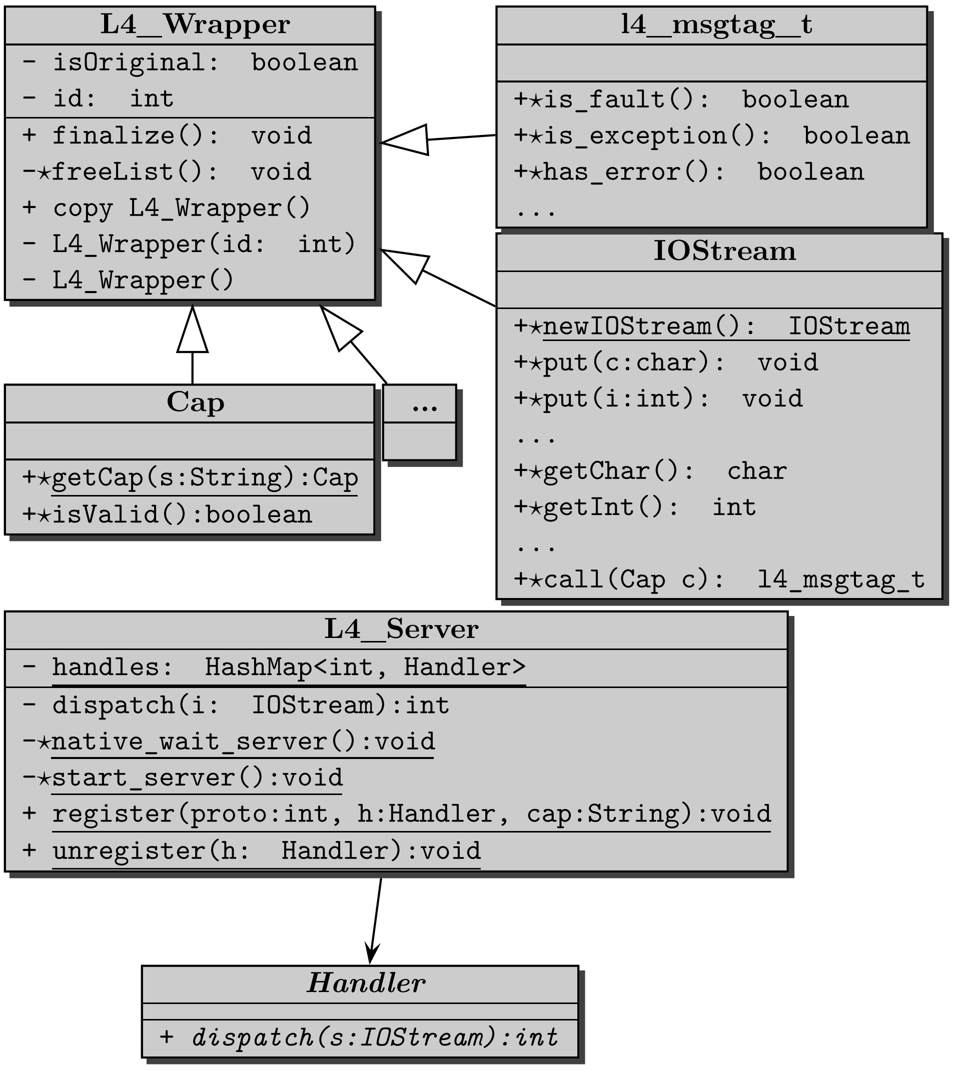

Figure 4.4: A suggestion for a better class structure with a cleaner interface

5 Evaluation

To show the performance of the J3 virtual machine on L4Re I choose SciMark2 as a benchmark. Because I also implemented the IPC interface for Java, I performed an IPC test as the second performance evaluation, to show the performance of IPC done between native programs and Java programs as well as between Java programs directly.

The following tests were executed using a Core i5 650 quad core processor, running at 3.2GHz and with a cache size of 4MB. The number of cores is not relevant for the SciMark2 test, because the benchmarks runs single threaded. Further, the machine used had 4GB of ram available, but not all of it was usable because a 32bit version of L4 was used.

5.1 SciMark2 benchmark

In the benchmarks I used the virtual machines J3 on Linux and L4 as well as OpenJDK with IcedTea and the SunJDK with the HotSpot virtual machine.

5.1.1 Problems with SciMark2 and J3

I originally planned to execute SciMark2 using the small and the large problem sets. But, although all the smaller test-samples used for testing J3 worked, the test initially led to a segfault. It was quite difficult to trace this problem, because it manifested itself in the form of a call to address 0 from within Just-in-Time compiled code. While I could not find the real origin of the bug, it turned out to be caused by using 64bit *long* values in Java. This bug did only manifest in the L4 version and was not present on the Linux version. The fix for this problem was to not use *System.currentTimeInMillis()* but instead use the *System.nanoTime()* function. The former function did the same but divided the result by 1000000 causing the aforementioned bug. I worked around this problem by first casting the returned number to double and then do the division. The unmodified SciMark would also have cast the resulting number to double anyways. However, because double is not an exact type, and the cast is done before the division, precision is lost. Because of this the results are not directly comparable to stock SciMark2 benchmarks, but should give a close indication. Also all the tests displayed were run using the same, modified, benchmark and are therefore comparable.

5.1.2 Results and interpretation

The results, shown in figure 5.1 on page 34, clearly show the performance of the J3 virtual machine is equal on both the Linux and the L4 versions. The performance difference of below 0.2% can be attributed to measurement errors. Also of interest is the comparison with the OpenJDK and SunJDK versions. Their performance is clearly superior to that

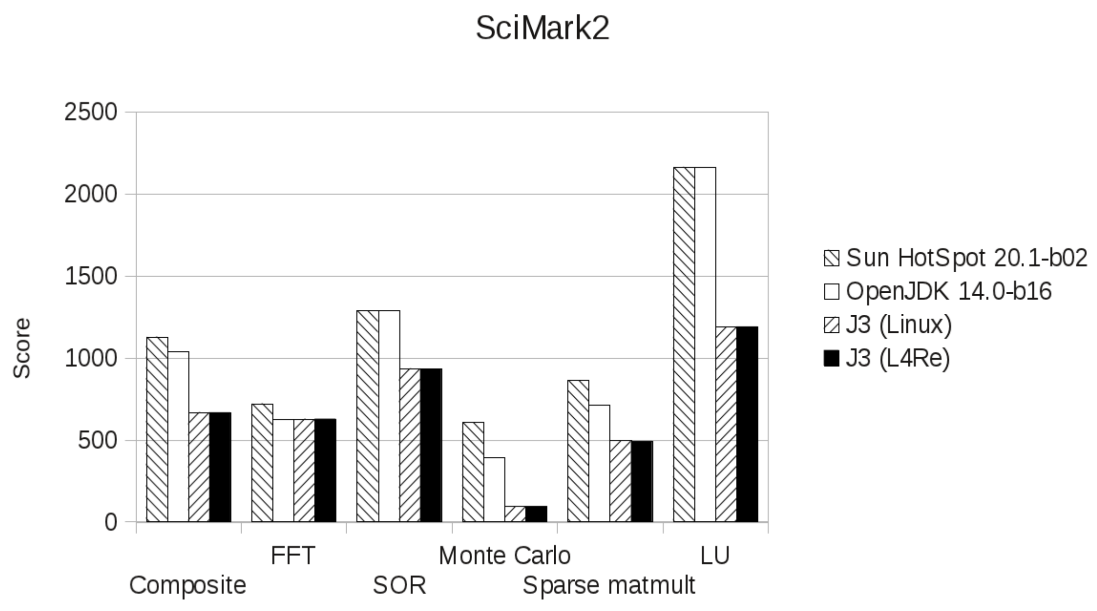

Figure 5.1: Results of the SciMark2 run, compared on various VMs

of J3. A fact that can probably be attributed to the design of J3. While OpenJDK and SunJDK directly translate to machine instructions J3 uses the intermediate representation of LLVM and translates to this first, the IR is then being translated to machine code. The difference to HotSpot can be attributed to the runtime optimizations of the HotSpot binary translator.

5.2 IPC tests

Next, I performed IPC tests to evaluate the performance of IPC servers and clients implemented in Java. The first test measured the time for several thousand IPCs executed in a loop and estimated the time per IPC by dividing the total runtime by the number of calls. To avoid side effects from the Just-in-Time compilation a first IPC is performed before the measurements and not included in the graphics. The program used for this was an adapted version of the ExampleCl called ExampleBench. The Java server and the native server were used unmodified and corresponding changes were made to the example client provided in the L4 examples. The measurements at first yielded extremely disappointing results. A Java client communicating with a native server only reached about 14k IPC/s and this number sharply declined after several hunderd iterations whereas a native server communicating with a native client showed about 1.2M IPC/s. Even more disappointing were the numbers obtained with both programs written in Java. Only about 50 IPC/s were possible.

This suggested that something was wrong with my implementation and I added more instrumentation to find the source of the slowdown. Because all time measurements that were available were only up to a millisecond resolution a more accurate method was needed. To get precise timings I implemented a call to *rdtsc* in java.lang.System class. This call only executes the RDTSC machine command to read the time stamp counter and returns its value to the caller. I instrumented the code with calls to read these time-stamps and at the end of each IPC the details were written to the screen and the serial console. In addition information about the minimal, maximal and average durations of the sections was provided at the end of the thousand IPCs.

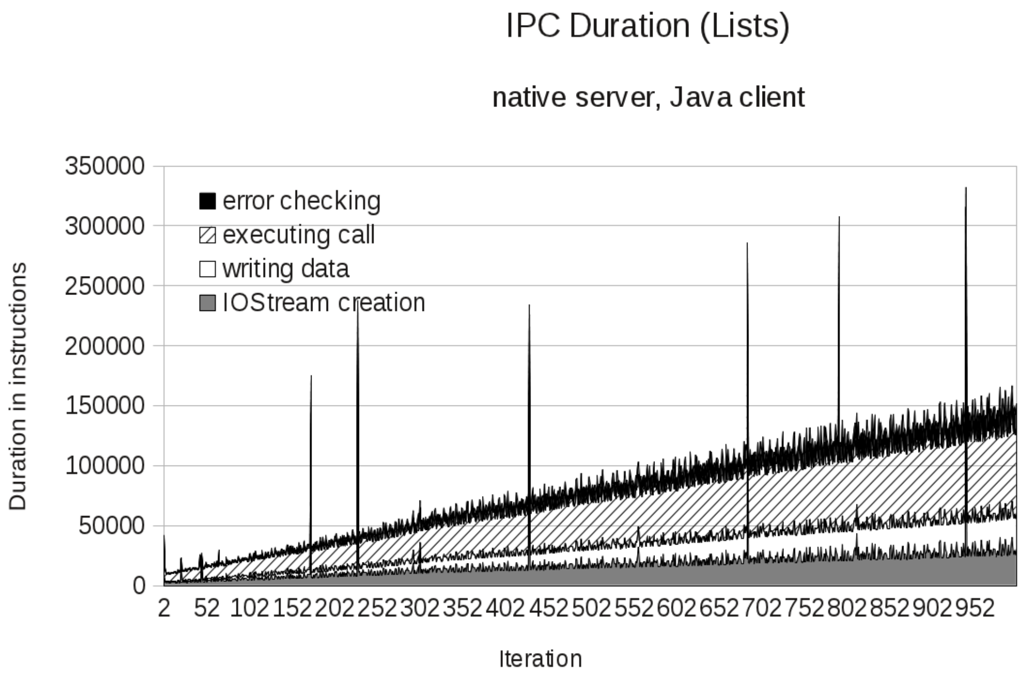

Figure 5.2: The performance over the individual iterations of the test, when native objects are stored in lists. The client is implemented in Java, the server is the native example server

The graph in figure 5.2 shows the results of this measurement over 1000 IPCs for the communication between a native server and the Java based client. The lines indicate the duration of the following parts of the client

- creation of the IPC stream
- writing 2 32 bit values to the stream
- executing the call to the server
- reading a 32 bit value from the stream, comparing it with the expected result and checking the return value of the call for errors

The y-axis shows the number of instructions that were executed for the individual step while the x-axis displays the individual measurements.

They are displayed in a stacked fashion, with the top line being equivalent to the total IPC time. In addition a measurement was taken calculating the overhead of a simple call of a native function from Java. This overhead is included in every function call. It amounts to a quite stable 248 instructions with the minimum being 156 instructions. The results were measured by calling the rdtsc function twice and subtracting its results. These values are not included in the graph.

The spikes that can be seen in figure 5.2 and also in figure 5.3 and that go up to an additional 150k instructions, are caused by dataspace mapping requests to moe.

The graph clearly shows a steep increase in execution time, starting at as low as about 12k cycles. This can easily be linked to the fact that native objects are stored in lists and searching lists as well as indexing into them is quite slow. In hindsight it was a bad decision to use lists in this place because the used inserting and element retrieval functions have a O(n) complexity and, because of this, are not suited for this application case. I found that dynamic arrays are much better in regard to append and indexing operations because they feature a O(1) lookup time and, as long as only appending is used, O(1) insertion. I adapted the implementation and used a default array size of 1024 elements, so that reallocation would not influence the measurements. The second graph,

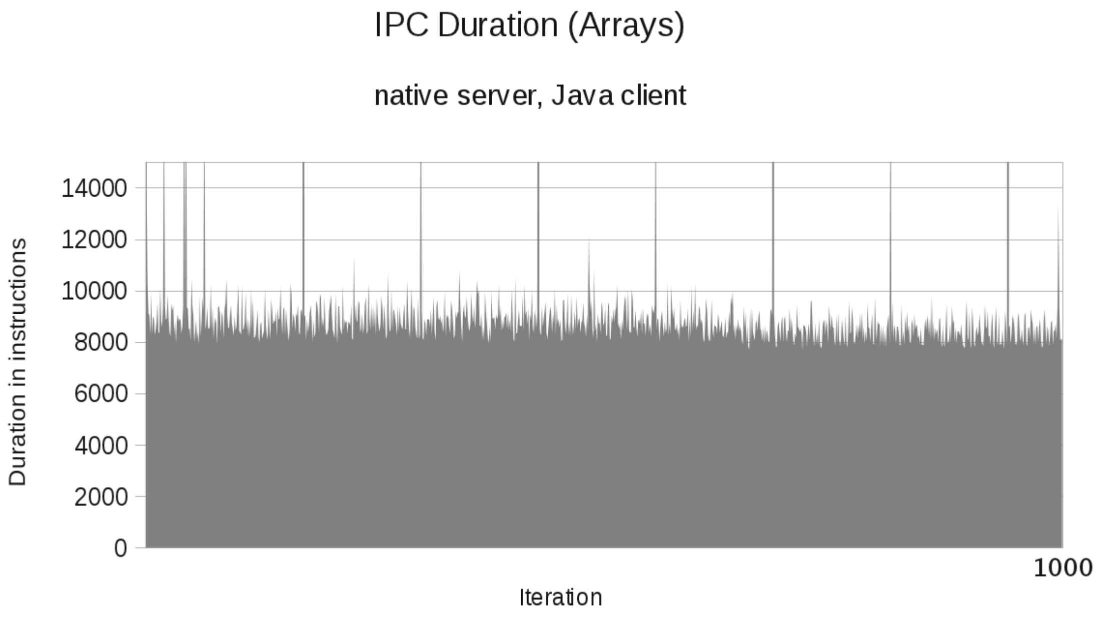

Figure 5.3: The same measurement method as in the previous figure, but using the modified native library with native objects stored in Arrays.

displayed in figure 5.3 was done using the modified array version of the client and shows a much better runtime behavior and the performance is more constant. The distribution of the execution time for the individual sections of the client was stable and the average of this distribution is shown in figure 5.4. Results for the IPC without the instrumentation can be seen in the chart in figure **??** at the end of this chapter.

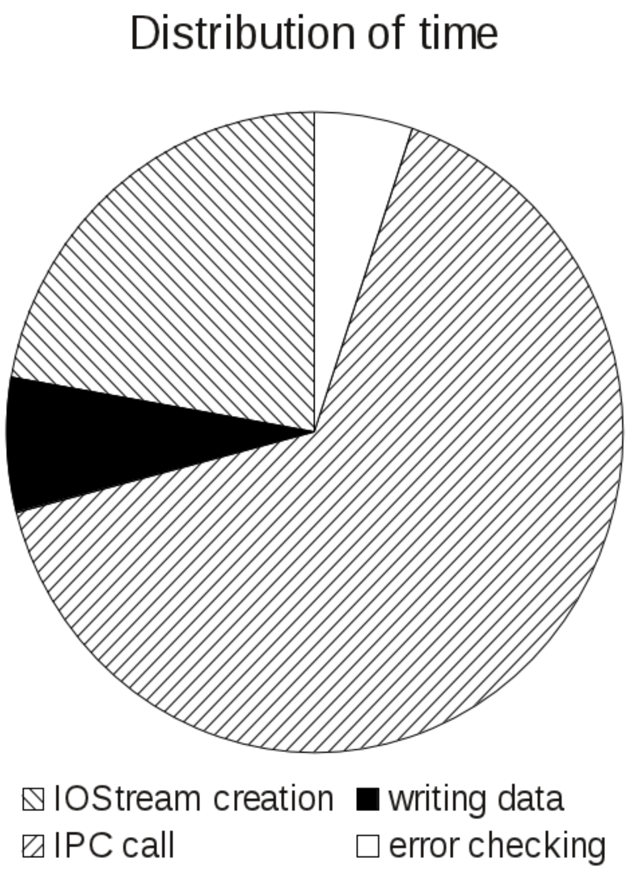

Figure 5.4: The time distribution of the individual parts of the Java IPC client. Averaged over the 1000 runs in figure 5.3

After the client's performance was closer to the native results and within the expected performance range I tested the Java server next. Its performance was recorded using the instrumented Java client and the results are displayed in the chart in figure 5.5. As can clearly be seen, the performance of the actual call is extremely bad with an average of about 80 million instructions spent there. This suggested that something in the server implementation was not performing as expected. Instrumentation of the call to the dispatch handler and several other parts showed good performance with less than 2000 instructions spent in the dispatch handler. This number is shown in the chart as *Server time*. I analyzed and instrumented the code further, which confirmed that the main point, where CPU instructions were wasted, was the busy waiting. The error here was trivial. In the loop where the variable of the other thread is checked, millions CPU cycles were burned because the thread would not yield its time. By inserting l4_thread_yield() into these loops I could reduce the cycle count of the call significantly. The combined benchmarks of the IPC are displayed in figure 5.6. While it is still up to 80% slower than native to native communication this will prove hard to avoid. Most of the time is spent in calls to native

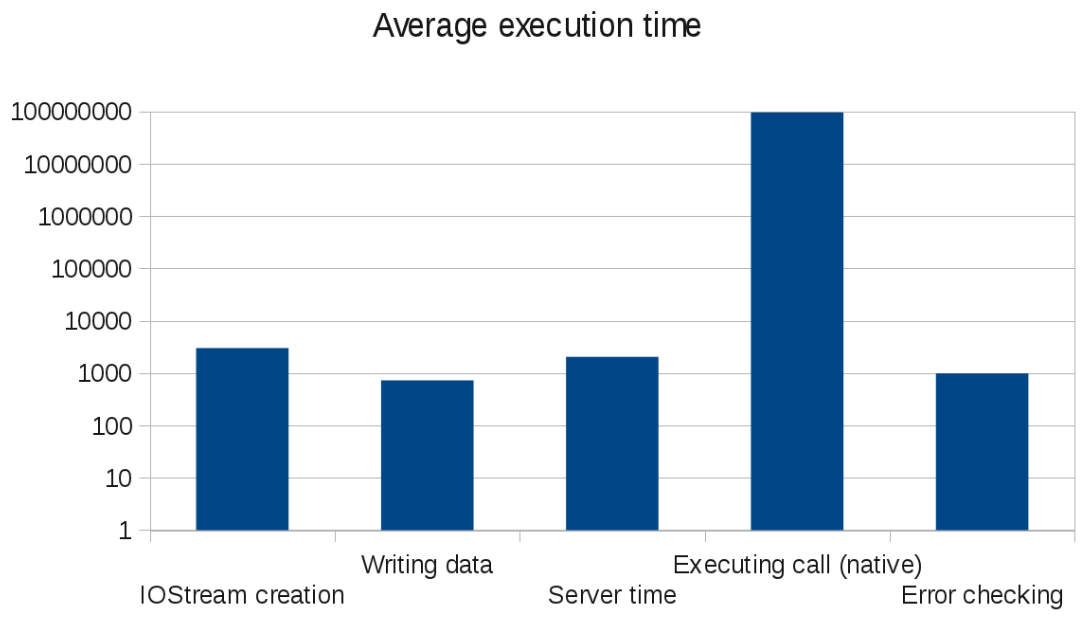

Figure 5.5: The time distribution of the individual parts of the Java IPC client for communication with a Java IPC sever. **Please note the logarithmic scale!**

functions. There is one possibility to use semaphores instead of busy waiting in the communication between threads, but because the contents of the IPC streams are destroyed by semaphores more work would need to be done to preserve the stream contents.

I also tested how the decision to use native objects and wrappers instead of Java objects affected performance. The method to do this was to move one of those functions into native code. I choose the msgtag_t object, because it is representative for the others, and reimplemented it as a Java object. I could not detect any performance gain by measuring this modified implementation. The reason for this observation is that a new object has to be created for each msgtag_t and the method to check for errors had to be recompiled by the JIT. While the implementation as a Java object saves one call to the native error checking routine, the overhead of JIT compiling the corresponding, more complex, Java routine outweighs this benefit. The performance of the two approaches was practically equal, with only a difference of 0.15 percent. The implementation that did not use a wrapper, but an object reimplemented in Java was only minimal faster here. This clearly shows that the approach that guarantees easier maintenance should be choosen, because there is no performance benefit in the more complex version that needs many objects reimplemented in Java.

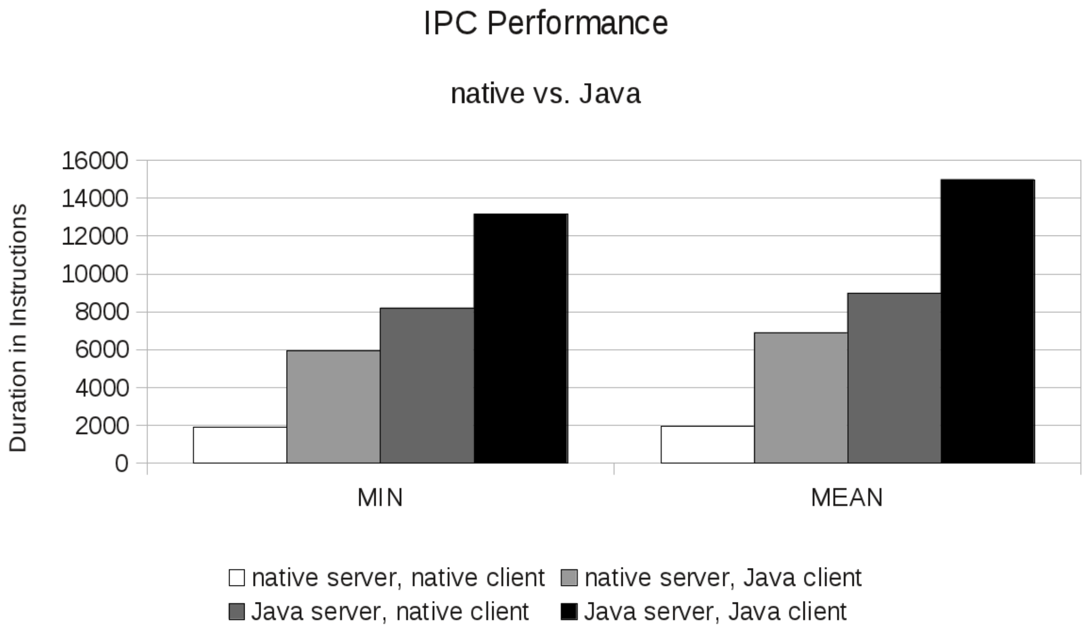

Figure 5.6: Chart comparing the performance of the Java to the native implementations

6 Conclusion and future work

6.1 Conclusion

As seen in the previous chapters J3 provides competitive performance for implementing L4 servers and clients in Java and for running generic code under L4. VMKit is also a good platform to more easily implement new virtual machines or to test new concepts for them. I gave a concept for simple implementation of the L4 interface in Java in chapter four, which should be easy to adopt to other virtual machines. The measurements taken in the evaluation chapter, especially after adapting the code, also showed that it is in general possible to write servers and clients in Java that communicate at a competitive speed.

As a side product of this work version, 2.8 of LLVM does now run on L4Re. This makes the LLI interpreter and Just-in-Time compiler available for use on L4 and enables execution of .bc compiled bytecode.

However, it must also be noted that there are still serious bugs remaining. The most prominent example of them is the late discovered bug with *long* variables in J3 as detailed on page 33. It is quite difficult to debug this problem, because a trace of the instructions from the actual manipulation of the integer to the call to address 0 leads through several sections of JIT compiled code and native code. With JIT code already difficult to debug under Linux with the support of gdb it becomes neigh impossible on L4Re without such a tool. Especially this bug prevents the unmodified execution of many Java classes on L4Re. I did further analyze this and found the bug to be related to another bug I encountered during benchmarking. The trigger of the *long* bug in the Java code is the creation of a character array that is used to decode long variables for printing. Arrays were also a problem during benchmarking, where the large problem size of SciMark2 could not be used because of this. The current version of the packages provided for L4Re is intended for Revision 40125 of the svn tree. Future versions might have changes that require the packages to be adapted to them.

6.2 Future work

There are several tasks that can be done to improve the quality of the current framework and that are still open. The problems I mentionied in the previous chapters, like the hack of the StackWalker on page 20, that runs over the top of the stack need further diagnosis and fixing. Also a more extensive test of the GNU Classpath would be advisable to find more still undetected bugs like the *long* problem described on page 33. It would also be nice to have other VMs ported to VMKit and to have them run on L4Re. But because documentation on how to implement such a VM is scarce and is mainly provided by the example of J3, it is quite a difficult task, because the interfaces are not clearly documented. Another interestng task would be to make MMTk available for LLVM as soon as it can be built using dragonegg instead of llvm-gcc, because this would enable simple implementation and test of different garbage collection strategies directly in Java.

Another area that would provide for future work is the complete implementation of the interface to L4Re in Java as was suggested in chapter 4 also using the improvements from the evaluation chapter.

Glossar

API	application programming interface
CLI	Common Language Infrastructure (.NET Framework)
CLR	Common Language Runtime
IPC	inter process communication
IR	intermediate representation (LLVM language)
JIT	Just-in-Time
L4Re	L4 Runtime Environment
LLVM	Low Level Virtual Machine
MRE	Machine Runtime Environment
VM	virtual machine
VMKit	Virtual Machine Kit
UTCB	user-level thread control block

Bibliography

[AGH05] Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005. 1

[Bar09] Edward Barrett. 3c - a JIT compiler with LLVM. Technical report, Bournemouth University, 2009. 9

[Bla04] Stephen M Blackburn. Oil and water? high performance garbage collection in java with mmtk. In *In ICSE 2004, 26th International Conference on Software Engineering*, pages 137–146, 2004. 7

[Fag05] Fabian Fagerholm. Perl 6 and the parrot virtual machine, 2005. 1

[fia11] The fiasco microkernel, September 2011. `http://os.inf.tu-dresden.de/fiasco/`. 5

[Geo09] N. Geoffray. Precise and efficient garbage collection in vmkit with mmtk, October 2009. Invited Talk at Apple for The LLVM Developer's Meeting http://llvm.org. 7

[gnu11] Gnu classpath - gnu project - free software foundation (fsf), September 2011. `www.gnu.org/software/classpath/`. 8

[GTJ+10] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Virtual Execution Environment Conference (VEE 2010)*, Pittsburgh, USA, March 2010. ACM Press. 1, 3, 4, 8

[HL04] Galen C. Hunt and James R. Larus. Singularity design motivation. TechReport MSR-TR-2004-105, Microsoft Research, 2004. 3

[Int06] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, June 2006. 1

[jav11] Open source web servers in java, September 2011. `java-source.net/open-source/web-servers`. 3

[jBS] Hans j. Boehm and Michael Spertus. Transparent programmer-directed garbage collection for c++. 7

[kaf04] Port of the java virtual machine kaffe to drops by using l4env, Juli 2004. `http://os.inf.tu-dresden.de/papers_ps/boettcher-beleg.pdf`. 9

[l4r11] L4re – the l4 runtime environment, September 2011. `http://os.inf.tu-dresden.de/L4Re/`. 2, 5

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. 6, 18

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`. 1, 6

[LBG02] Chris Lattner, Misha Brukman, and Brian Gaeke. Jello: a retargetable just-in-time compiler for llvm bytecode, 2002. 1

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. 8

[Lie95] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995. 5

[llv11a] http://llvm.org/pubs/, 2011. 9

[llv11b] The llvm compiler infrastructure project, September 2011. `www.llvm.org`. 6

[LW09] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM. 5

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. 1

[MWG00] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the common language runtime, 2000. 1

[oEEE95] Institute of Electrical and Inc. Electronic Engineers. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York, NY, 1995. 5

[Pig08] Alessandro Pignotti. An efficient actionscript 3.0 just-in-time compiler implementation, 2008. Bachelor Thesis, Universita degli Studi di Pisa. 9

[pyt08] Administration and debugging shell for l4env, May 2008. `http://os.inf.tu-dresden.de/papers_ps/pohl-beleg.pdf`. 10

[zli11] zlib home site, September 2011. `http://zlib.net/`. 5